ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Java Security: Architecture and Primitives

Alessandro Buldini

alessandro.buldini@unibo.it

Java

- First appeared 29th May 1995.
- High-level **OOP language** developed by Sun Microsystems.
- **Platform-independent**: Java Virtual Machines are built for most operating systems meaning Java programs can run pretty much everywhere without changing the code.
- **Robust, reliable, and <u>safe</u>**: Java is a <u>statically typed</u> language that provides extensive <u>compile-time checking</u>, followed by a second level of <u>run-time checking</u>. There are no explicit programmer-defined pointer data types, **<u>no pointer arithmetic</u>**, and **<u>automatic garbage collection</u>** [1].

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Safety vs. Security

Safety and Security are two related but distinct concepts:

- **Safety** focuses on preventing **accidental failures** that could harm the application or the system the program is run on. Examples are *garbage collection, static typing, exceptions handling, thread synchronization, impossibility to handle pointers, strong encapsulation (via private, public, protected).*
- **Security**, on the other hand, focuses on protecting from **intentional attacks** by malicious actors.

Java Security Architecture

Java Development Kit, JDK, defines a set of high-level APIs spanning over major security areas, including [2]:

- **Cryptography** (**Hash, Digital Signatures, Ciphers, MACS, PRNGs**, …)
- Public Key Infrastructure (X.509 certs, CRLs, path validation, …)
- Authentication (secure login modules for LDAP, Kerberos, Windows NT, Unix, …)
- Secure communication (TLS, Datagram-TLS, SSL, …)
- Access control (permissions, security policies, AC enforcement, …)

These APIs allow developers to integrate security into their application code.

Java Security Overview

Security in Java is provided via several modules that contain security API [2]:

- `java.base`: foundational security for Java Standard Edition. Includes: **`java.security`**, **`javax.crypto`**, **`javax.net.ssl`**, **`javax.security.auth`**.
- `java.smartcardio`, provides smartcard secure I/O APIs.
- `java.jartool`, provides tools to sign JAR files.
- […]

Java API

Java API are designed around the following principles [2]:

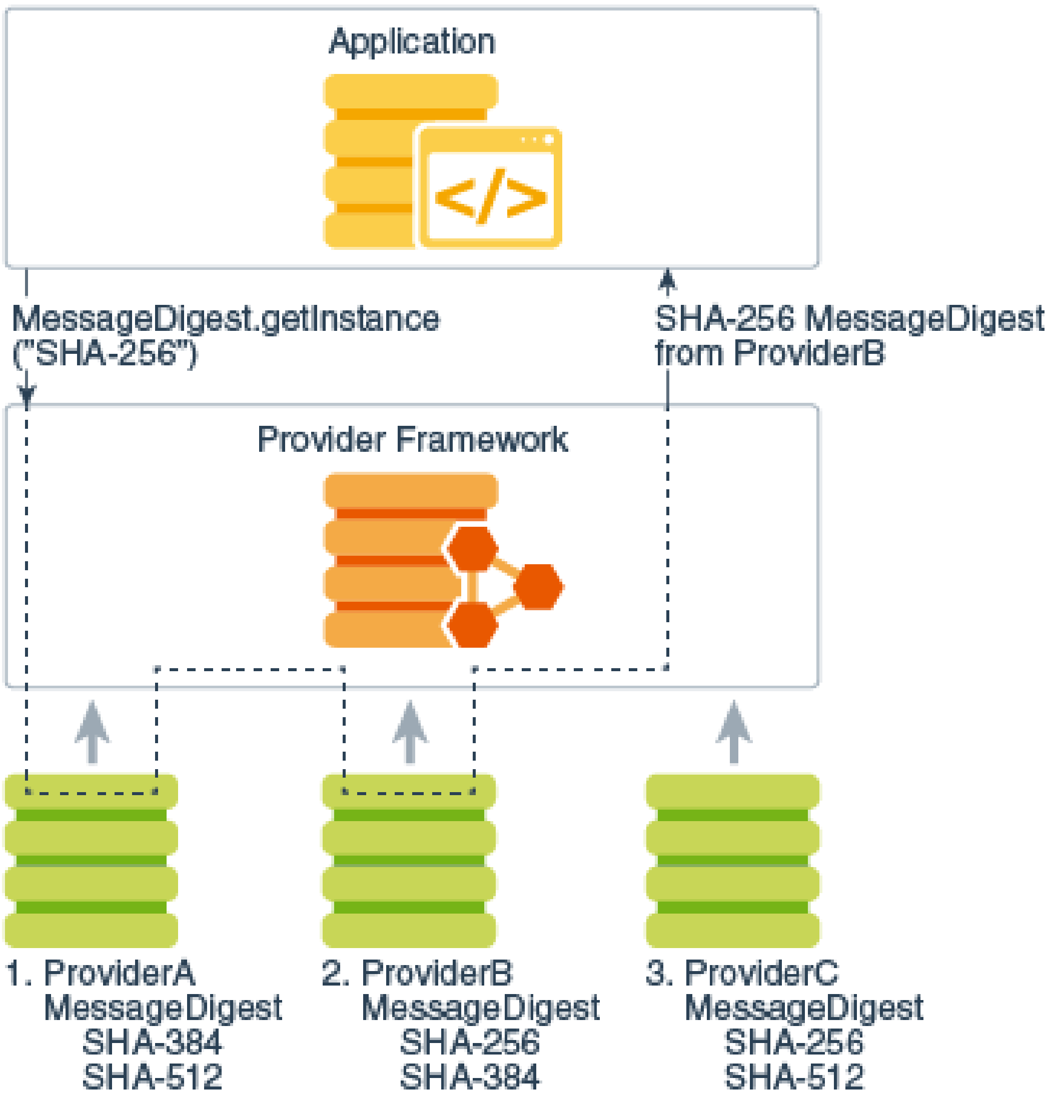
- **Implementation independence**. Applications do not need to implement security themselves. They do so by requesting services from **Cryptographic Service Providers (CSP)** which are plugged into the JDK via a standard interface.
- **Implementation interoperability**. Providers are interoperable across applications: a program is not bound to a specific provider if it does not rely on default values from it.
- **Algorithm extensibility**: some applications may rely on emerging standards not yet implemented. The JDK supports the installation of custom providers that implement such services.

Cryptographic Service Providers [3]

- Every CSP refers to a package or set of packages that implement one or more cryptographic services, such as **digital signature algorithms, message digest algorithms**, and key conversion services.
- Providers may be updated transparently to the application, for example when faster or more secure versions are available.
- **Implementation interoperability** means that various implementations can work with each other, use each other's keys, or verify each other's signatures.

Why multiple providers?

- I may want to implement my own provider or use my favorite one (bouncy-castle, IBM's IBMJCEPlus, Microsoft's MSCAPI …) instead of the default Oracle implementation.
- Providers have priorities.
- Some providers may perform cryptographic operations in **software**; others may perform the operations on a **hardware cryptographic accelerator**.



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Achieving interoperability (1)

- We likely won't create a custom provider ourselves, so we're not really interested in how to implement providers.

- From a software engineering perspective, though, it's very interesting to see how interoperability and modularity is achieved in OOP.

- Algorithm independence is achieved by defining types of **cryptographic services called *engines*** and defining classes that provide the functionality of these services. These classes are called **engine classes**, and examples are the MessageDigest, Signature, and Cipher classes.

Achieving interoperability (2)

- **Engine classes** are the ***abstract*** classes we will be working with. They **extend** a *root abstract class* which defines the behavior of the cryptographic component. These behavior-defining classes are called **Service Provider Interface (SPI)**.

- The engine class implementing a hashing algorithm, for example, must have a function to produce a digest:
  `public byte[] digest();`
- Within the `digest` function, the customizable behavior is achieved by calling a method **defined in the SPI** parent class and **implemented in the provider**. The signature of the called method will be:
  `protected abstract byte[] engineDigest();`

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

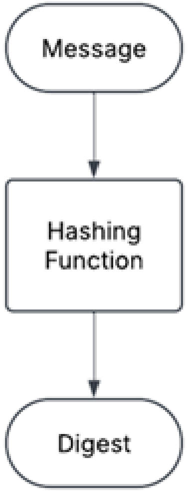
Achieving interoperability (3)

```
Completes the hash computation by performing final operations such as padding. The digest is reset
after this call is made.

Returns: the array of bytes for the resulting hash value.

public byte[] digest() {
    /* Resetting is the responsibility of implementors. */
    byte[] result = engineDigest();
    state = INITIAL;
    return result;
}
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Hashing – Providing *integrity* (1)

- To avoid modifications of the information in the unsafe channel, we need a **hashing function** that takes in **input a string of information of any length** and outputs a **unique fingerprint of a fixed length**.
- Inputting the same string of information into a hashing function always outputs the same fingerprint.
- The modification of a single bit must completely change the output fingerprint.

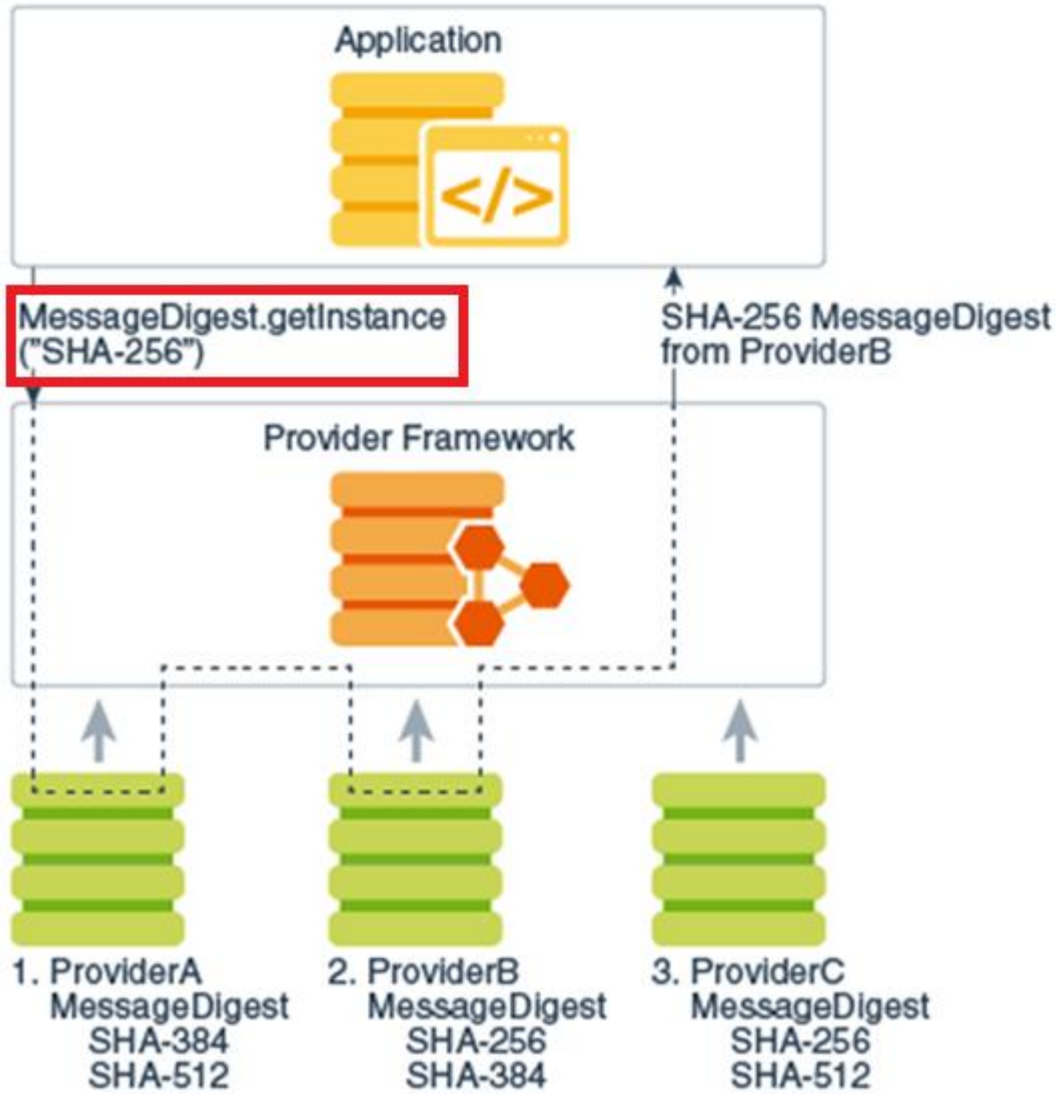


Hashing – Providing *integrity* (2)

Hashing in Java (1) [4]

- To use hashes (*/digests/fingerprints*) in our Java application, we have to employ the `MessageDigest` class.

- **<u>Obtainable, like every other cryptographic object granted by CSPs, exclusively via the</u>** `getInstance(String name)` **<u>static method</u>**.



- `public static MessageDigest(String algorithm);`
- *`MessageDigest md = MessageDigest.getInstance("SHA-256")`*

Hashing in Java (2) [4]

The `MessageDigest` class provides three methods (and many overloading implementations) to manage digest creation:

- `public final void update(ByteBuffer input);`
  Updates the input buffer by initializing it or appending `input` to it.
- `public final void reset();`
  Resets the input buffer.
- `public final byte[] digest();`
  Computes and returns the hash/digest/fingerprint of the input. <u>It automatically resets the input buffer</u>.

Hashing Example (1) - Explanation

```
try {

    String message = "Sicurezza dell'Informazione";                    // Message we want to produce the digest of
    byte[] messageBytes = message.getBytes();                          // Message converted to a byte array
    ByteBuffer buf = ByteBuffer.wrap(messageBytes);
    MessageDigest md = MessageDigest.getInstance( algorithm: "SHA-256");  // Retrieve a MessageProvider instance from the first provider available
    md.update(buf);                                                    // Updates the input buffer of the engine class with the message
    byte[] digest = md.digest();                                       // Computes and returns the digest
    String hexDigest = HexFormat.of().formatHex(digest);               // Converts the digest to hex format
    // md.reset();                                                     // Useless here since input buffer already reset within the md.digest() function

    System.out.println(hexDigest);                                     // "add4b4c68d5febb2cce9675b19e17c7aa9a4897ce9e73f32665e47abc6260642"

} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
```

Hashing Example (2) - Update

```java
try {

    MessageDigest md = MessageDigest.getInstance( algorithm: "SHA-256");  // Retrieve a MessageProvider instance from the first provider available
    md.update("Sicurezza ".getBytes());                                  // Here we're using an overloaded method that takes in input a byte array
    md.update("dell'".getBytes());
    md.update("informazione".getBytes());

    byte[] digest = md.digest();
    System.out.println(HexFormat.of().formatHex(digest));                // "add4b4c68d5febb2cce9675b19e17c7aa9a4897ce9e73f32665e47abc6260642"

} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
```

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Hashing Example (3) – Empty Buffer

```java
try {

    String emptyMessage = "";
    byte[] messageBytes = emptyMessage.getBytes();
    MessageDigest md = MessageDigest.getInstance( algorithm: "SHA-256");
    md.update(messageBytes);                                        // Here we're using an overloaded method that takes in input a byte array

    byte[] digest = md.digest();
    System.out.println(HexFormat.of().formatHex(digest));           // "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"

    byte[] digest2 = md.digest();                                   // No need to reset input buf. Done by previous md.digest() call
    System.out.println(HexFormat.of().formatHex(digest2));          // "e3b0c44298fc1c149afbf4c8996fb92427ae41e4649b934ca495991b7852b855"

} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
```

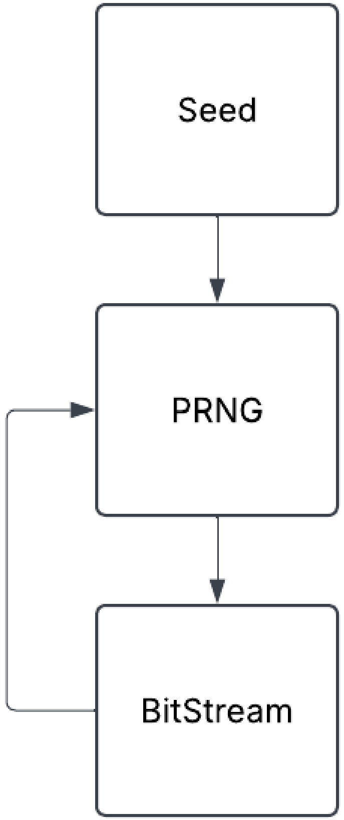
Hashing Example (4) – Reset

```java
try {

    MessageDigest md = MessageDigest.getInstance( algorithm: "SHA-256");          // Retrieve a MessageProvider instance from the first provider available
    md.update("AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA".getBytes());
    md.reset();                                                                    // We reset the input buffer
    md.update("Sicurezza dell'informazione".getBytes());                         // Here we're using an overloaded method that takes in input a byte array

    byte[] digest = md.digest();                                                   // Digest computed over "Sicurezza dell'informazione"
    System.out.println(HexFormat.of().formatHex(digest));                          // "add4b4c68d5febb2cce9675b19e17c7aa9a4897ce9e73f32665e47abc6260642"

} catch (NoSuchAlgorithmException e) {
    e.printStackTrace();
}
```

Pseudo Random Number Generators

Random Number Generators are algorithms that generate a random bit stream for which the possibility of guessing the next bit is $\frac{1}{2} + \epsilon$ where ϵ is negligible. There are two types of RNGs:

- **PRNG**: sample randomness from a cyclic group, hashing functions, or ciphers.
- **TRNG**: sample randomness from a native source of randomness.



Non-Secure PRNG example

Cyclic groups can be used to generate randomness. Given a prime p in the form $p - 1 = q \cdot k$, there exists at least one generator g of a cyclic group (of cardinality $p - 1$).

In the following example, $\boldsymbol{p = 7}$, $\mathrm{p} - 1 = 6 = 3 \cdot 2$, $\boldsymbol{g = 3}$.

A number n is a generator of $p - 1$ if $g^{\frac{p-1}{q}} \neq 1 \bmod p$, and $g^{\frac{p-1}{k}} \neq 1 \bmod p$.

$3^1 \bmod 7 = 3 \bmod 7 \quad = 3$

$3^2 \bmod 7 = 9 \bmod 7 \quad = 2$

$3^3 \bmod 7 = 27 \bmod 7 \quad = 6$

$3^4 \bmod 7 = 81 \bmod 7 \quad = 4$

$3^5 \bmod 7 = 243 \bmod 7 \quad = 5$

$3^6 \bmod 7 = 729 \bmod 7 \quad = 1$

$3^7 \bmod 7 = 2187 \bmod 7 \quad = 3$

$3^8 \bmod 7 = 6561 \bmod 7 \quad = 2$

…

RNGs in Java (1) [5]

- Cryptographically secure Random Number Generators are available through the `SecureRandom` class.

- They mainly rely on PRNGs, with some exceptions.
  - Algorithm **`SHA1PRNG`** leverages cryptographically secure hashing function properties of randomness to generate a random bit.
  - Algorithm **`DRBG`** (Deterministic Random Bit Generator) leverages hashing functions, ciphers, or elliptic curve cryptography **and sample randomness through an entropy source**.

All implementations are shown in [6].

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

RNGs in Java (2) [5]

- The `SecureRandom` class overrides the `java.util.Random` class. It's a fairly simple class:
- To retrieve an instance of the SecureRandom object:
  `public static SecureRandom getInstance(String algorithm);`
  `SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");`
- `public void setSeed(long seed);`
  sets the seed in the `SecureRandom` PRNG.
- Overridden `java.util.Random` methods:
  - `public int nextInt();` Securely generates a random integer.
  - `public int nextBoolean();` Securely generates a random boolean.
  - `public int nextFloat();` Securely generates a random float.
  - […]

RNGs Example (1)

```java
try {

    SecureRandom sr = SecureRandom.getInstance(algorithm: "SHA1PRNG");  // Retrieves an instance of SHA1PRNG from the first available provider.
    sr.setSeed(31337);                                                  // Sets the seed within the PRNG instance.
    int randomInt = sr.nextInt();                                       // Samples an integer from the PRNG.
    System.out.println(randomInt);                                      // -548606946
    float randomFloat = sr.nextFloat();                                 // Samples a float from the PRNG.
    System.out.println(randomFloat);                                    // 0.49130863

} catch (Exception e) {
    e.printStackTrace();
}
```

RNGs Example (2) – Different instances, same values

```java
try {
    SecureRandom sr = SecureRandom.getInstance( algorithm: "SHA1PRNG");
    sr.setSeed(31337);
    System.out.println(sr.nextInt());  //  -548606946
    sr.setSeed(1337);
    System.out.println(sr.nextInt());  //    68561509

    SecureRandom sr2 = SecureRandom.getInstance( algorithm: "SHA1PRNG");
    sr2.setSeed(1337);
    System.out.println(sr2.nextInt()); // -1596841925
    System.out.println(sr2.nextInt()); // -1446375891

    SecureRandom sr3 = SecureRandom.getInstance( algorithm: "SHA1PRNG");
    sr3.setSeed(1337);
    System.out.println(sr3.nextInt()); // -1596841925
    System.out.println(sr3.nextInt()); // -1446375891

} catch (Exception e) {
    e.printStackTrace();
}
```

RNGs Example (3) – Different instances, different values

```java
try {
    SecureRandom sr = SecureRandom.getInstance( algorithm: "DRBG");
    sr.setSeed(31337);
    System.out.println(sr.nextInt());  // -1095615748
    sr.setSeed(1337);
    System.out.println(sr.nextInt());  //  -197071089

    SecureRandom sr2 = SecureRandom.getInstance( algorithm: "DRBG");
    sr2.setSeed(1337);
    System.out.println(sr2.nextInt()); // -1891210154
    System.out.println(sr2.nextInt()); //  1110222379

    SecureRandom sr3 = SecureRandom.getInstance( algorithm: "DRBG");
    sr3.setSeed(1337);
    System.out.println(sr3.nextInt()); //  1787292078
    System.out.println(sr3.nextInt()); //    -7632326

} catch (Exception e) {
    e.printStackTrace();
}
```

Symmetric Ciphers

Symmetric ciphers are ciphers in which the keys to encrypt and decrypt data are identical, similar, or easily computable the one from the other.

The participants must preemptively agree on the key.

Two types of symmetric ciphers exist:

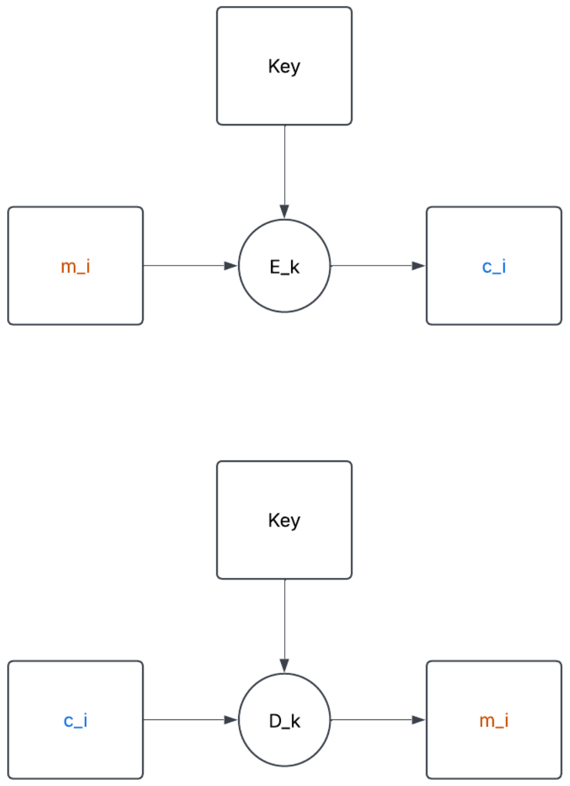
- Block ciphers.
- Stream ciphers.

Block Ciphers

<u>In block ciphers the message is subdivided into blocks</u> of *n* bits. A **transformation** is then applied to each block both when **encrypting** and when **decrypting**.

Allows for different transformations (mode of operations):

- Electronic Codebook – **ECB** (few cases only).
- Cipher Block Chaining – **CBC**.
- Output Feedback – **OFB**.
- Cipher Feedback – **CFB**.
- Counter – **CTR**.
- Galois-Counter Mode – **GCM** (if you need black-box approach, always use this one*).
- Counter with CBC-MAC Mode – **CCM**.



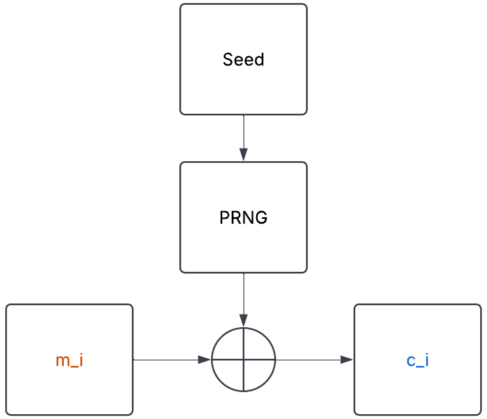
*pay attention to slide Disclaimer later.

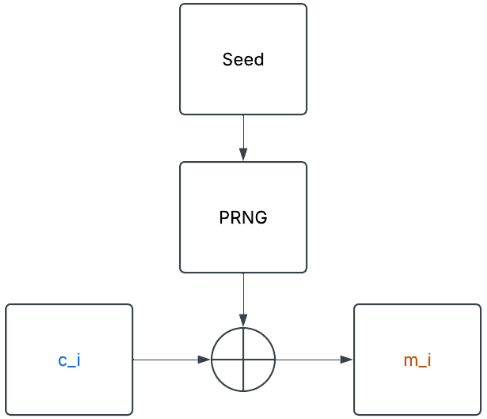
Stream Ciphers

Stream ciphers <u>leverage the XOR operation between plaintext bits and random bits</u> obtained from PRNGs to produce ciphertext.

Two types of Stream Ciphers exist:

- <u>Synchronous</u> Stream Ciphers only depend on the keystream. (*image on the side is a simplification of a Synchronous Stream cipher*).
- <u>Self-Synchronizing</u> Stream Ciphers depend on both the anteceding ciphertext and the keystream.





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Ciphers in Java (1) [7]

- One can use either <u>stream or block ciphers</u> in java by leveraging the engine class `Cipher`.
- An instance of `Cipher` is obtainable through once again through

```
public static Cipher getInstance(String transformation);
Cipher cipher = Cipher.getInstance("AES");        // block
Cipher cipher = Cipher.getInstance("ChaCha20"); // stream
```

- <u>**For block ciphers only**, to specify the **Mode of Operations** and the **padding algorithm**</u>, the input string can contain additional information:

```
Cipher c = Cipher.getInstance("AES/ECB/PKCS5Padding");
```

Modes of Operations (1) [7]

- Depending on the mode of operations, some additional information might be required:
  - **ECB**: no additional information.
  - **CBC**: requires an Initialization Vector.
  - **OFB**: requires an Initialization Vector.
  - **CFB**: requires an Initialization Vector.
  - **CTR**: requires an Initialization Vector and No padding.
  - **CCM**: not implemented in the standard library, we would need to add BouncyCastle dependency through maven or gradle, then add the provider via *`Security.addProvider(new BouncyCastleProvider());`*
  - **GCM**: requires an Initialization Vector, the MAC dimension, and No padding.

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Disclaimer

- Initialization Vector management is paramount to provide security.
- **For GCM, the cipher class should be re-initialized with a different IV every time we need to encrypt data with the same key [7]. Failure to do so allows forgery attacks.**
- For **CCM**, failure to re-initialize the IV compromises authentication of encrypted data.
- For **CTR, OFB, CFB**, where the IV is essential to generate a keystream, failure to re-initialize the IV compromises the privacy, allowing an easier retrieval of the plaintext.

- In the following examples, we'll supply a fixed-seed PRNG to always be able to compute the same data and produce the same results. In real life scenarios, other than re-initializing the cipher class for correct IV and nonce management, **always make sure the PRNG never ever produces the same data**. **Do not set a fixed seed**.

Modes of Operations (2) [7]

- DISCLAIMER: Make sure you read correctly the previous slide.
- First, create a PRNG and fill a byte array:

```
SecureRandom sr = SecureRandom.getInstance("SHA1PRNG");
sr.setSeed(1337);
byte[] iv = new byte[16];
sr.nextBytes(iv);
```

- To provide an Initialization Vector:

```
IvParameterSpec spec = new IvParameterSpec(iv);
```

- For GCM specifically, we also need to provide the length in bit of the MAC*.

```
int macLen = 128;
GCMParameterSpec spec = new GCMParameterSpec(macLen, iv);
```

*We'll be back on this in a bit.

Generating private keys (1)

- Of course, to use ciphers, we need cryptographic key material. To generate keys, we need a `KeyGenerator` object.
- With no one's surprise, available via:

```
public static KeyGenerator getInstance(String algorithm);
KeyGenerator kg = KeyGenerator.getInstance("AES");
```

- Initialize the object using:

```
public void init(int keySize);
kg.init(256);
```

- Obtain a secret key:

```
public SecretKey generateKey();
SecretKey sk = kg.generateKey();
```

Disclaimer

- In the following examples, we will be using an overloaded method to initialize the `KeyGenerator` object:
- `public void init(int keySize, SecureRandom sr);`

```
SecureRandom sr = SecureRandom("SHA1PRNG");
sr.setSeed(1337);
kg.init(256, sr);
```

- In the following examples, we'll supply a fixed-seed PRNG to always be able to compute the same data and produce the same results. In real life scenarios, when generating keys, **always make sure the PRNG never ever produces the same data**. Do not use PRNGs with a fixed set seed.

Ciphers in Java (2) [7]

- Once we have set up all the necessary parameters, we can initialize the Cipher class:
  ECB: `public final void init(int opmode, Key secretKey);`
  Others: `public final void init(int opmode, Key secretKey, AlgorithmParameterSpec spec);`
- *`cipher.init(Cipher.ENCRYPT_MODE, sk, spec);`*
  `cipher.init(Cipher.DECRYPT_MODE, sk, spec);`
- In the signature of the `init` method, `AlgorithmParameterSpec` is a superclass of both `IVParameterSpec`, and `GCMParameterSpec`.
- Depending on the mode of operation, every time we need to update the IV, we need to create new specs and re-call the `init` method.

Ciphers in Java (3) [7]

- Finally, when the cipher object has been initialized, we can start encrypting or decrypting.
- For <u>one-shot encryption/decryption</u> we use:
  `public final byte[] doFinal(byte[] input);`
- For <u>multi-part encryption/decryption</u>, assuming we have `n` parts*:
  ```
  public final byte[] update(byte[] input); // for n-1 parts
  public final byte[] doFinal(byte[] input);// for last part
  ```
- <u>Do not use the overloaded method</u> ~~`public void doFinal();`~~
  <u>This method is not fully interoperable between modes of operations.</u>

*`Update` returns a `byte[]` for compatibility with stream ciphers. Results before calling `doFinal` might not be consistent (or might be null).

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

AEAD

- The **GCM** and **CCM** operating modes belong to the category of Authenticated Encryption with Additional Data ciphers (AEAD) as they both leverage a MAC to grant integrity. These ciphers not only grant the authenticity of the **ciphertext**, but also of some **additional plaintext data**.

- **GCM** is encrypt-then-mac (integrity of the ciphertext).
- **CCM** is mac-then-encrypt (integrity of the plaintext). Chosen for historical reasons, decrypting and verifying via MtE can cause a wide range of attacks. Still used in IoT.

- Due to how the class `Cipher` calls engine methods, there is a strict limitation on the expressiveness of the `doFinal()` overloaded method in representing AEAD modes. Do not call the overloaded variant with no arguments.

AEAD in Java

- There is no standard CCM implementation for Java.
- About **GCM**, it **handles every aspect of the authentication transparently**. The only distinction with other operating modes is the `parameterSpec` passed to the `init` function.

- To authenticate additional (non-encrypted) data, we can use the following function:
  `public final void updateAAD(byte[] src);`
- Of course, this function must be called both when encrypting, and decrypting, in the same order.

Stream Ciphers

- To employ stream ciphers like **ChaCha20** we must supply additional parameters exactly as we did for modes of operations.
- **ChaCha20**, for example, must be supplied with a **nonce** and a **counter** variable.

```
byte[] nonce = new byte[12];
sr.nextBytes(nonce);
int counter = sr.nextInt();
ChaCha20ParameterSpec spec = new
                    ChaCha20ParameterSpec(nonce, counter);
```

- *Additionally, to fully enable the capabilities of stream ciphers it's possible to wrap them into `CipherInputStream` and `CipherOutputStream` objects.*

Ciphers Example (1) – Block Ciphers, ECB

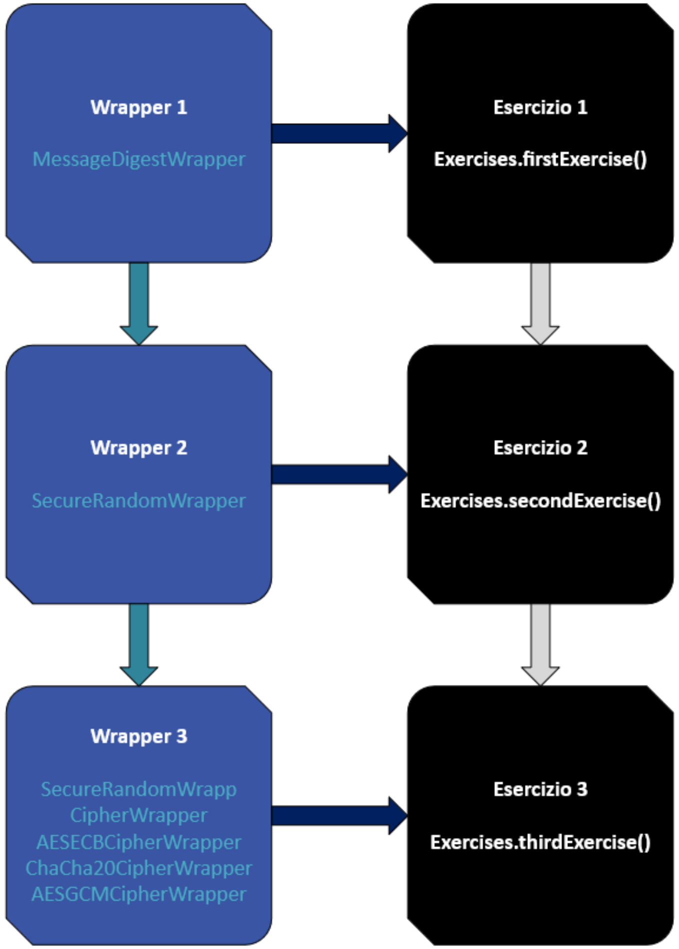
```
try {

    SecureRandom sr = SecureRandom.getInstance( algorithm: "SHA1PRNG");            // For the example we need a PRNG to always produce the same output.
    sr.setSeed(1337);                                                               // In real life scenarios, this is something you DON'T want.

    KeyGenerator keyGen = KeyGenerator.getInstance( algorithm: "AES");              // Engine class to generate simmetric keys.
    keyGen.init( keysize: 256, sr);                                                 // Initialize it with key size and the PRNG.
    SecretKey secretKey = keyGen.generateKey();                                     // Compute a secret key.
    Cipher cipher = Cipher.getInstance( transformation: "AES/ECB/PKCS5Padding");    // Retrieve a cipher object from the first available provider.

    /* Encryption */
    cipher.init(Cipher.ENCRYPT_MODE, secretKey);                                    // Initialize the cipher with parameters.
    byte[] plaintextBytes = "Sicurezza dell'informazione".getBytes();               // Convert plaintext to byte array.
    byte[] encrypted = cipher.doFinal(plaintextBytes);                              // Produce the ciphertext.
    String encodedCiphertext = HexFormat.of().formatHex(encrypted);                 // (Encode it for fancy display).
    System.out.println("Encrypted bytes: " + encodedCiphertext);                    // "753d67dd8340ddb80146fec3a37f513b6c04b299149794334914a590809299f6"

    /* Decryption */
    cipher.init(Cipher.DECRYPT_MODE, secretKey);                                    // (Re-)Initialize the cipher object.
    byte[] decryptedBytes = cipher.doFinal(encrypted);                              // Produce the plaintext byte array.
    String decrypted = new String(decryptedBytes);                                  // Encode it in a string.
    System.out.println("Decrypted bytes: " + decrypted);                            // "Sicurezza dell'informazione"

} catch (Exception e) {
    e.printStackTrace();
}
```

Ciphers Example (2) – Stream Ciphers, ChaCha20

```java
try {

    SecureRandom sr = SecureRandom.getInstance( algorithm: "SHA1PRNG");                 // For the example we need a PRNG to always produce the same output.
    sr.setSeed(1337);                                                                     // In real life scenarios, this is something you DON'T want.

    byte[] nonce = new byte[12];                                                         // Declare the nonce.
    sr.nextBytes(nonce);                                                                 // Fill the nonce with (pseudo) random bytes.
    int counter = sr.nextInt();                                                          // Counter initialization variable.

    ChaCha20ParameterSpec paramSpec = new ChaCha20ParameterSpec(nonce, counter);         // Create ChaCha parameter object.
    Cipher cipher = Cipher.getInstance( transformation: "CHACHA20");                    // Initialize the cipher object with ChaCha20 algorithm.

    KeyGenerator keyGen = KeyGenerator.getInstance( algorithm: "CHACHA20");              // Engine class to generate simmetric keys.
    keyGen.init( keysize: 256, sr);                                                      // Initialize it with key size and the PRNG.
    SecretKey secretKey = keyGen.generateKey();                                          // Compute a secret key.

    /* Encryption */
    cipher.init(Cipher.ENCRYPT_MODE, secretKey, paramSpec);                              // Initialize the cipher with parameters.
    byte[] plaintextBytes = "Sicurezza dell'informazione".getBytes();                    // Convert plaintext to byte array.
    byte[] encrypted = cipher.doFinal(plaintextBytes);                                   // Produce the ciphertext.
    String encodedCiphertext = HexFormat.of().formatHex(encrypted);                     // (Encode it for fancy display)
    System.out.println("Encrypted bytes: " + encodedCiphertext);                         // "4580e2ecc921a638caf49e3041b3f9b3d9904a2cb7ef282fc83dcd"

    /* Decryption */
    cipher.init(Cipher.DECRYPT_MODE, secretKey, paramSpec);                              // (Re-)Initialize the cipher object.
    byte[] decryptedBytes = cipher.doFinal(encrypted);                                   // Produce the plaintext byte array.
    String decrypted = new String(decryptedBytes);                                       // Encode it in a string.
    System.out.println("Decrypted bytes: " + decrypted);                                 // "Sicurezza dell'informazione"

} catch (Exception e) {
    e.printStackTrace();
}
```

Ciphers Example (3) – AEAD

```java
try {

    SecureRandom sr = SecureRandom.getInstance( algorithm: "SHA1PRNG");               // For the example we need a PRNG to always produce the same output.
    sr.setSeed(1337);                                                                 // In real life scenarios, this is something you DON'T want.

    byte[] iv = new byte[16];                                                         // Declare the initialization vector.
    sr.nextBytes(iv);                                                                 // Fill the IV with (pseudo) random bytes.
    GCMParameterSpec spec = new GCMParameterSpec( tLen: 128, iv);                     // Create parameter object according to the operating mode.
    Cipher cipher = Cipher.getInstance( transformation: "AES/GCM/NoPadding");         // Retrieve a cipher object from the first available provider.

    KeyGenerator keyGen = KeyGenerator.getInstance( algorithm: "AES");                // Engine class to generate simmetric keys.
    keyGen.init( keysize: 256, sr);                                                   // Initialize it with key size and the PRNG.
    SecretKey secretKey = keyGen.generateKey();                                       // Compute a secret key.

    /* Encryption */
    cipher.init(Cipher.ENCRYPT_MODE, secretKey, spec);                                // Initialize the cipher with parameters.
    byte[] plaintextBytes1 = "Sicurezza".getBytes();                                  // Convert plaintext to byte array.
    byte[] plaintextBytes2 = " dell'informazione".getBytes();                         // Convert plaintext to byte array.
    cipher.updateAAD("Professoressa: Rebecca Montanari".getBytes());
    cipher.update(plaintextBytes1);                                                   // Produce the ciphertext.
    byte[] encrypted = cipher.doFinal(plaintextBytes2);
    String encodedCiphertext =  HexFormat.of().formatHex(encrypted);                  // (Encode it for fancy display).
    System.out.println("Encrypted bytes: " + encodedCiphertext);                      // "76acb7a0de15f47e42f23f53dcb5b9b37daceca12c40176960cb21a8442374957f04d2d54255b49d3a150f"

    /* Decryption */
    cipher.init(Cipher.DECRYPT_MODE, secretKey, spec);                                // (Re-)Initialize the cipher object.
    cipher.updateAAD("Professoressa: Rebecca Montanari".getBytes());
    byte[] decryptedBytes = cipher.doFinal(encrypted);                                // Produce the plaintext byte array.
    String decrypted = new String(decryptedBytes);                                    // Encode it in a string.
    System.out.println("Decrypted bytes: " + decrypted);                              // "Sicurezza dell'informazione"

} catch (Exception e) {
    e.printStackTrace();
}
```


Wrapper 1
MessageDigestWrapper
Esercizio 1
Exercises.firstExercise()
Wrapper 2
SecureRandomWrapper
Esercizio 2
Exercises.secondExercise()
Wrapper 3
SecureRandomWrapp
CipherWrapper
AESECBCipherWrapper
ChaCha20CipherWrapper
AESGCMCipherWrapper
Esercizio 3
Exercises.thirdExercise()

https://github.com/alebldn/Esercitazione2603

Bibliografia

- [1]: Introduction to Java - https://www.oracle.com/java/technologies/introduction-to-java.html
- [2]: Java Security Overview - https://docs.oracle.com/en/java/javase/23/security/java-security-overview1.html
- [3]: Java Cryptograpy Architecture - https://docs.oracle.com/en/java/javase/23/security/java-cryptography-architecture-jca-reference-guide.html
- [4]: MessageDigest - https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/security/MessageDigest.html
- [5]: SecureRandom - https://docs.oracle.com/en/java/javase/23/docs/api/java.base/java/security/SecureRandom.html
- [6]: SecureRandom implementations - https://docs.oracle.com/en/java/javase/11/security/oracle-providers.html#GUID-9DC4ADD5-6D01-4B2E-9E85-B88E3BEE7453
- [7]: Cipher - https://docs.oracle.com/en/java/javase/23/docs/api/java.base/javax/crypto/Cipher.html
- [8]: StreamCipher code variation- https://docs.oracle.com/en/java/javase/23/security/java-cryptography-architecture-jca-reference-guide.html#GUID-C0283BC0-8B88-480D-82B1-7B01EAC3D8DF

ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Alessandro Buldini

alessandro.buldini@unibo.it